The SRE's Crystal Ball

Predicting Performance with Queues and USL

Aravindh Sampathkumar

Booking.com

October 09, 2025

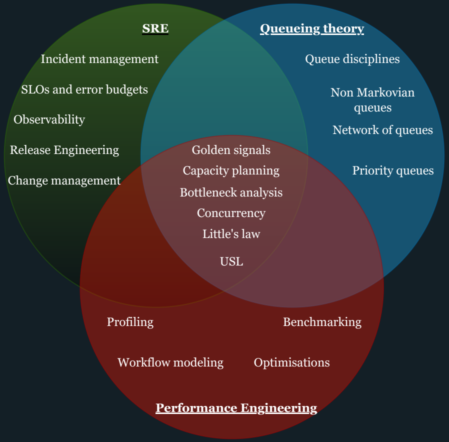
About Me

Aravindh Sampathkumar
Site Reliability Engineer @ Booking.com

Background

- High Performance Computing (HPC)
- Storage Systems
- Performance Engineering

The big picture



SRE

Incident management

SLOs and error budgets

Observability

Release Engineering

Change management

Queueing theory

Queue disciplines

Non Markovian queues

Network of queues

Priority queues

Golden signals

Capacity planning

Bottleneck analysis

Concurrency

Little's law

USL

Profiling

Benchmarking

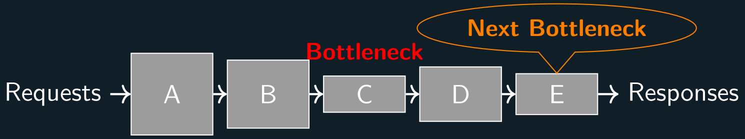
Workflow modeling

Optimisations

Performance Engineering

The big picture

- Let’s have some fun.
- Aimed at inspiring you, my peer practitioners to apply these concepts in ways I haven’t thought of.
- A system/service is a chain of bottlenecks - removing one reveals the next.



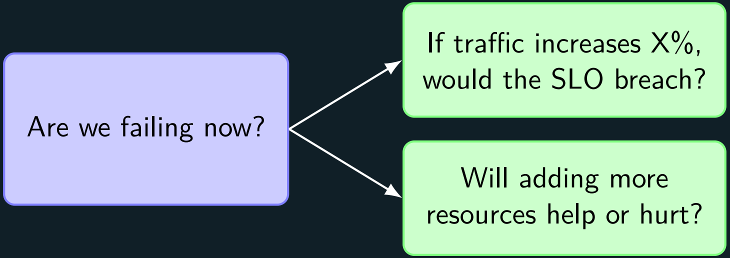
Credits

- Dr. Neil J. Gunther authored Universal Scalability Law and taught GCAP workshop.
- Stefan Moeding maintains the R package that I use in Demo.
- Phil Larson who mentored and introduced me to applied queueing theory.

Why bother?

Most SLOs are reactive alarms

- **Availability SLO Miss:** *”We’re down.”*
- **Latency SLO Miss:** *”We’re slow.”*



Are we failing now?

If traffic increases X%, would the SLO breach?

Will adding more resources help or hurt?

Move from firefights to preventing *some* issues entirely — think in **queues** and **scalability models**.

Agenda

- Thinking in queues
- Golden signals through the lens of queueing theory
- Predicting scalability with Universal Scalability Law(USL)
- See it in action

The Birth of Queueing Theory

Figure: A. K. Erlang (1878-1929) Source: Wikipedia

- In the early 1900s, Danish mathematician **Agner Krarup Erlang** had to figure out how many telephone circuits were needed to handle a given number of calls without excessive waiting or dropped connections.
- His work created queueing theory — the mathematical study of waiting lines (or queues).

Thinking in queues

- Queues are everywhere - Loadbalancers, connection pooling, Jira boards..

Thinking in queues

Thinking in queues

- Queueing systems are non-linear - It is not intuitive. Double the instance count and you can serve double the load right?
- Queues occur even if there is enough average capacity - Grocery store
  - High variance in arrvivals
  - High variance in service times

Thinking in queues

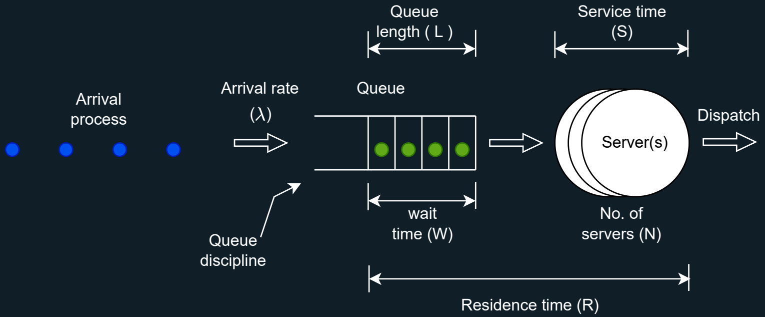


Figure: A simple queue.

Thinking in queues - Terminology

Queue length (L)	Number of requests in the system or queue(L_q)
Arrival Rate (λ)	The rate at which requests enter the system (e.g., requests/sec). aka ”demand” or ”load”.
Service Time (S)	The time required for a single server to process one request.
Waiting Time (W)	The time a request spends waiting in the queue before its service begins.
Response Time aka Latency(R)	The total time a request is in the system. The sum of waiting and service time ($R = W + S$).
Utilization (ρ)	The fraction of time a server is busy.
Number of servers (N)	The number of servers in the system(node/cpu/thread/instance).
Throughput (X)	Completion rate of requests. In a stable system, ($X = \lambda$).

Golden signals 👓 through queueing theory

- Latency $=$ Service time $+$ Waiting time
  - Utilisation drives waiting time
  - Coefficient of variation is a leading indicator
- Traffic as Arrival rate(λ): The demand driver
- Errors as symptoms of overload
- Saturation as high Utilisation(ρ): The harbinger of Doom

Little's Law and Related Formulae

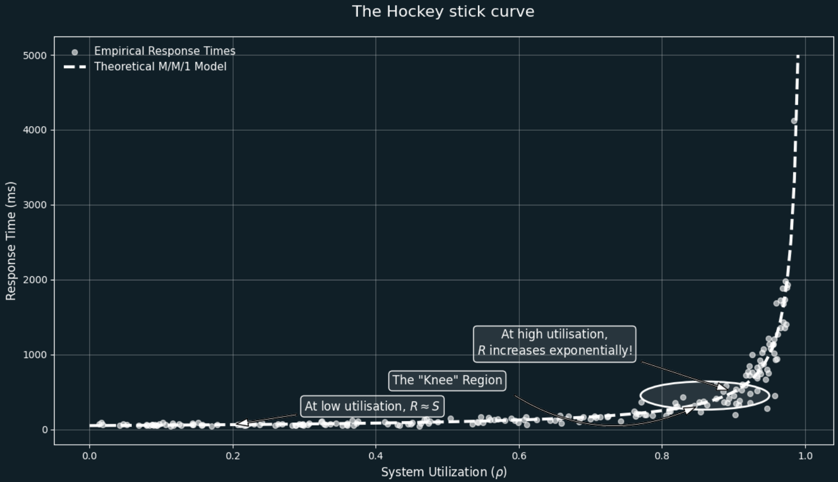
$$L = \lambda R$$

$$\rho = \lambda S$$

or

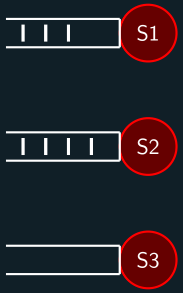
$$\rho = \frac{\lambda S}{N}$$

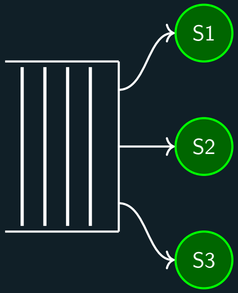
Utilisation curve and the "Knee"



A little intuition

Which design performs better?





Applied Theory: The Case of the Sluggish API

🔔 An API endpoint (‘/api/v1/resource‘) is timing out. Latency has spiked!

Applied Theory: The Case of the Sluggish API

🔔 An API endpoint (‘/api/v1/resource‘) is timing out. Latency has spiked!

What do we know?

- **Arrival Rate (λ):** 180 requests/second.
- **Avg. Service Time (S):** 50 milliseconds per request.
- **Number of Servers (N):** 4 API server pods.

Applied Theory: The Case of the Sluggish API

🔔 An API endpoint (‘/api/v1/resource‘) is timing out. Latency has spiked!

What do we know?

- **Arrival Rate (λ):** 180 requests/second.
- **Avg. Service Time (S):** 50 milliseconds per request.
- **Number of Servers (N):** 4 API server pods.

- *"Should we scale up to 8 pods? "*
- *"Would it be enough?"*

Applied Theory: The Case of the Sluggish API

$$\rho = \frac{\lambda S}{N} = \frac{\text{Arrival Rate} \times \text{Service Time}}{\text{Number of Servers}}$$

$$\rho = \frac{180 \text{ req/s} \times 0.05 \text{ s/req}}{4 \text{ pods}} = \frac{9}{4} = \mathbf{2.25}$$

Applied Theory: The Case of the Sluggish API

$$\rho = \frac{\lambda S}{N} = \frac{\text{Arrival Rate} \times \text{Service Time}}{\text{Number of Servers}}$$
$$\rho = \frac{180 \text{ req/s} \times 0.05 \text{ s/req}}{4 \text{ pods}} = \frac{9}{4} = \mathbf{2.25}$$

- Utilisation ($\rho > 1.0$ or 100%) $\Rightarrow$ the system is unstable. In other words, the queue is growing infinitely.
- Increase N to 8 pods? $\Rightarrow$ ρ will compute to 1.125 (still > 1.0).
- Let's increase N to at least 10 pods ($\rho = 0.9$) to also accommodate variance.

USL

I promised a crystal ball. Lets do some predictions.

USL

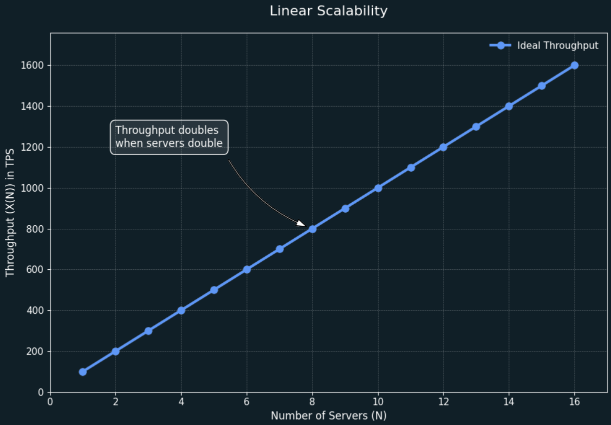
Scalability
A mathematical function of being able to perform more work (RPS, TPS etc) while work per server[1] remains constant and the number of servers increases.[*]

Universal Scalability Law(USL)
A formal definition of scalability.

[1] (node/instance, cpu/thread etc)
[*] Improvised from a definition by Dr. Neil J Gunther

USL - Linear Scalability

The ideal condition. We all want that right?

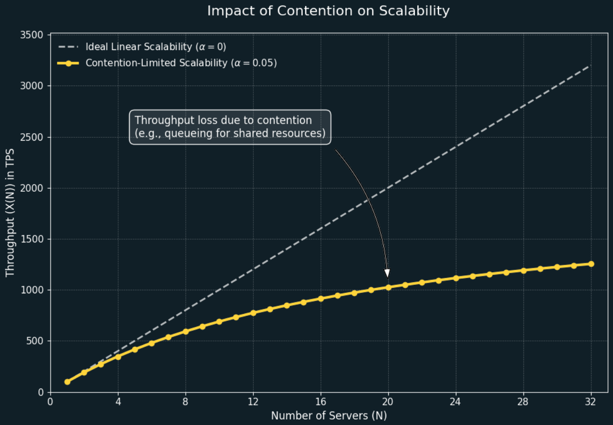


$$X(N) = \frac{\gamma N}{1}$$

Where:

- $X(N)$ is the throughput with N servers.
- γ is the ideal throughput of a single server.
- N is the number of servers.

USL - Scalability villain no.1 - Contention

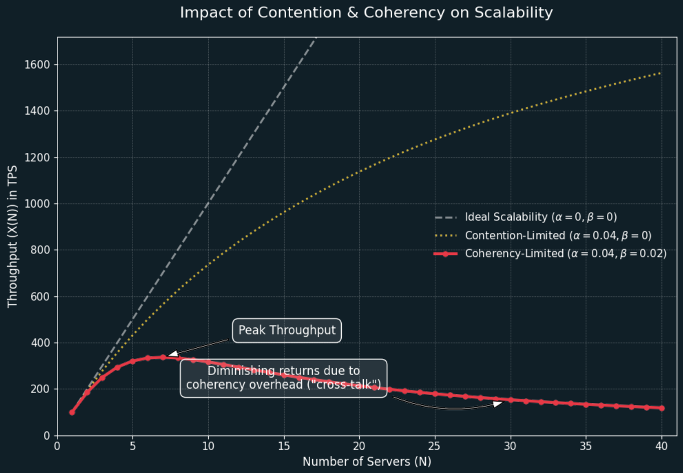


$$X(N) = \frac{\gamma N}{1 + \alpha(N-1)}$$

Where:

- α represents contention.
- contention is non-parallelisable serialised work.

USL - Scalability villain no.2 - Coherency (a.k.a. Crosstalk)



USL

$$X(N) = \frac{\gamma N}{1 + \alpha(N-1) + \beta N(N-1)}$$

Where:

- γ represents ideal single server throughput.
- α represents contention.
- β represents coherency penalty.

USL - What do we do with it?

Model system/scalability (obviously).

1. Apply observed system/service behaviour metrics.
2. Estimate α, β, and γ using non-linear least squares regression.
3. Interpret the results to know scalability limits and improve bottlenecks.

USL

Demo
Source at https://github.com/aravindhsampath/srecon25-usl

USL - Limits

- Noisy data - real-world data is quite noisy(network jitter, OS scheduler, GC pauses etc.).
- Distributed systems - harder to model aggregate functions(/order depends on 10 microservices).
- Asynchronous and event driven architecture pattern is hard to model.
- Coherency factor(β) assumes 1-1 coordination.
- Systems dont always behave the same way at scale. E.g switch to a different algorithm or co-ordinate in batches etc.
- Noisy neighbours - multi-tenancy.
- Rate limits and throttling at dependencies.

Common pitfalls - USL

Keep these in mind while working with USL.

- Garbage In Garbage Out - be diligent about noise. Noise will have non linear effects.
- Over-extrapolation - Dont forecast too far(new constraints emerge at scale). 2x is the rule of thumb.
- Max throughput if often NOT the goal - max throughput @desired latency is.

Key takeaways

- Start thinking in queues. Enrich your monitoring - capture service times and waiting times in histograms, queue lengths.
- Use Little's law for napkin math of fundamental relationship between Utilisation, Throughput/arrival rate, and response time.
- Use USL as a diagnostic compass to identify coarse contention and coherance penalties - rethink design choices to achieve practical goals.
- Keep queing theory wisdom in mind while designing services:
  - Favour pooled resources - single queue and a shared pool of workers.
  - Attack variability - Use caching strategies, optimisations for slower code paths etc to reduce service time variability.
  - Heavy-tailed service times? consider priority queues or size based routing.

References and further reading

- Dr. Neil J Gunther's page on Scalability and USL.
- Baron Schwartz's The essential Guide to QueueingTheory.
- Neil J. Gunther and Stefan Moeding's USL R package
- Eben Freeman's LISA 17 talk on Queueing theory in practice
- Kavya Joshi's talk on Applied Performance Theory.

Thank You!

Questions?

Get in Touch

- ✉ : aravindh at fastmail.com
- in : linkedin.com/in/aravindhsampath
- : github.com/aravindhsampath
- Bluesky : aravindh.net

Thank You!

Questions?

Get in Touch

- ✉ : aravindh at fastmail.com
- in : linkedin.com/in/aravindhsampath
- : github.com/aravindhsampath
- Bluesky : aravindh.net